System design: Yelp

Suresh Podeti
6 min read · Oct 6, 2023

Photo by mostafa meraji on Unsplash

Introduction

Yelp is a one-stop platform for consumers to discover, connect, and transact with local businesses. With it, users can join a waitlist, make a reservation, schedule an appointment, or purchase goods easily. Yelp also provides information, photos, and reviews about local businesses. User can also upload their opinions in the form of text, pictures, or ratings.

Services based on **proximity servers** are helpful in finding nearby attractions such as restaurants, theaters, or recreational sites. Designing such a system is challenging because we have to **efficiently find all the possible places in a given radius with minimum latency**. This means that we have to narrow down all the locations in the world, which could be in the billions, and only pinpoint the relevant one

Requirements

Functional

- **Add**: Business owners can add their places on the platform.
- **Search**: The users should be able to search for nearby places or places of interest based on their GPS location (longitude, latitude) and/or the name of a place.
- **Feedback**: The users should be able to add a review about a place. The review can consist of images, text, and a rating.

Non-functional

- **High availability**: The system should be highly available to the users.
- **Scalability**: The system should be able to scale up and down, depending on the number of requests. The number of requests can vary depending on the time and number of days. For example, there are usually more searches made at lunchtime than at midnight. Similarly, during tourist season, our system will receive more requests as compared to in other months of the year.
- **Consistency**: The system should be consistent for the users. All the users should have a consistent view of the data regarding places, reviews, and images.
- **Performance**: Upon searching, the system should respond with suggestions with minimal latency.

Storage schema

Place	
Place_ID:	INT
Name_of_place:	VARCHAR(256)
Description_of_place:	VARCHAR(1000)
Category:	VARCHAR
Latitude:	DECIMAL
Longitude:	DECIMAL
Photo_ID:	INT
Rating:	DECIMAL

Photo	
Photo_ID:	INT
Place_ID:	INT
Photo_path:	VARCHAR(256)

Reviews	
Review_ID:	INT
Place_ID:	INT
User_ID:	INT
Review_description:	VARCHAR
Rating:	INT

Users	
User_ID:	INT
User_name:	VARCHAR

Fig 1.0: Storage schema needed for storing places, photos, reviews, and users

Design

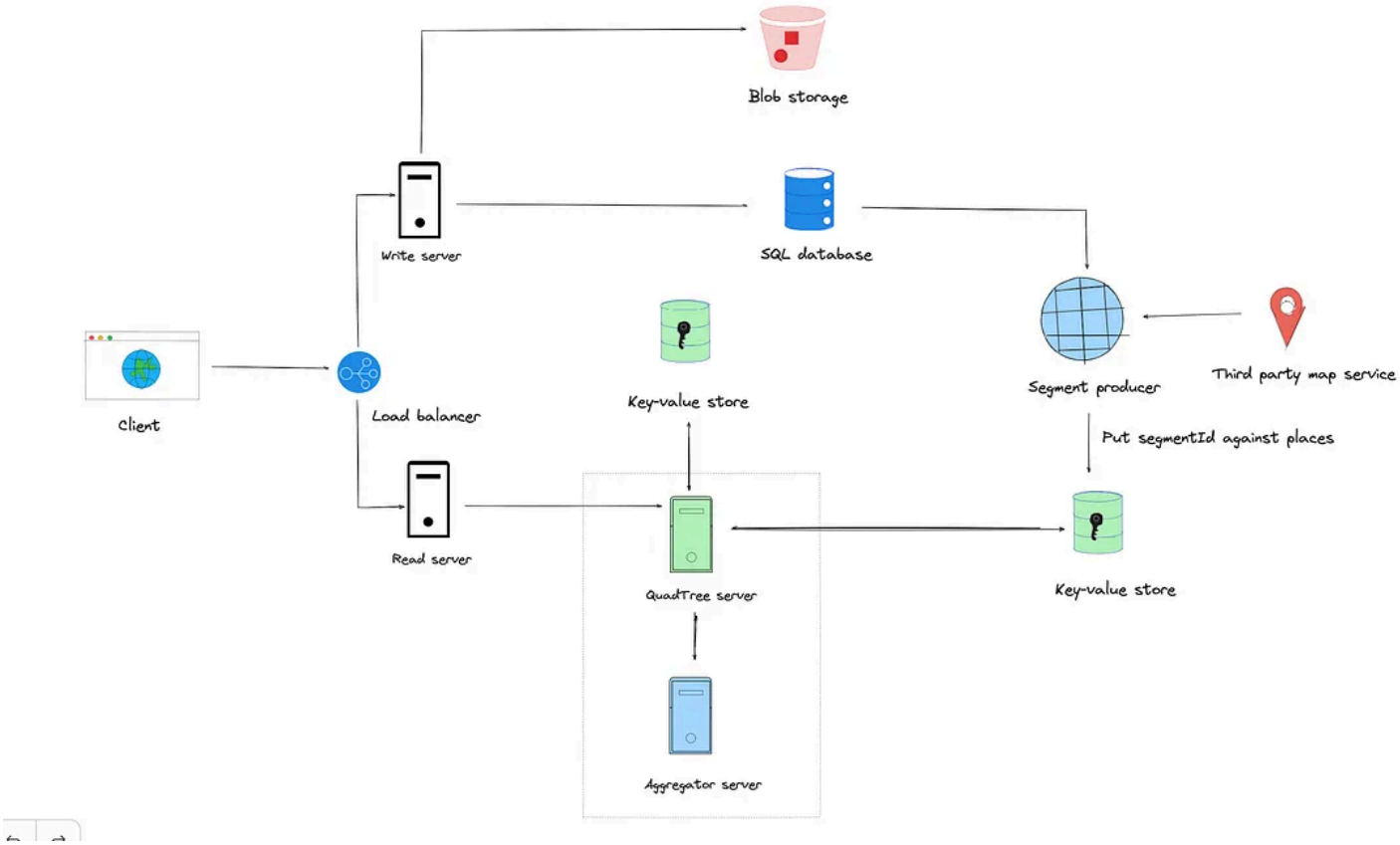

Fig 2.0: High level design of Yelp

Components

- **Segment producer:** This component is responsible for communicating with the third-party world map data services (for example, Google Maps). It takes up that data and divides the world into smaller regions called segments. The segment producer helps us narrow down the number of places to be searched.

- **QuadTree servers:** These are a set of servers that have trees that contain the places in the segments. A QuadTree server finds a list of places based on the

given radius and the user's provided location and returns that list to the user. This component mainly aids the search functionality.

- **Aggregators:** The QuadTrees accumulate all the places and send them to the aggregators. Then, the aggregators aggregate the results and return the search result to the user.
- **Read servers:** We use a set of read servers that we use to handle all the read requests. Since we have more read requests, it's efficient to separate these requests from the write requests. Each read server directs the search requests to the QuadTrees' servers and returns the results to the user.
- **Write server:** We use a set of write servers to handle all the write requests. Each write server handles the write requests of the user and updates the storage accordingly.
- **Storage:** We'll use two types of storage to fulfill our diverse needs.

SQL Database: Our system will have different tables like Users, Place, Reviews, and Photos. The data in the tables are inherently relational and structured. We need to perform queries like places a user visited, reviews they have added, etc. It is easy to perform such queries in a SQL-based database. We also want all users to have a consistent view of the data, and SQL-based databases are better suited for such use cases.

Key-value stores: We'll need to fetch the places in a segment efficiently. For that, we store the list of places against a segmentID in a key-value store (`segmentId: [place1, place2, place3]`) to minimize searching time. We also store the quadTree information in the key-value store, by storing the quadTree against a uniqueID.

Workflow

Searching a place: The load balancers route read requests to the read servers upon receiving them. The read servers direct them to the QuadTree servers to find all the places that fall within the given radius. The QuadTree servers then send the results to the aggregators to refine them and send them to the user.

Adding a place or feedback: The load balancers route the write requests to the write servers upon receiving them. Depending on the provided content, meaning the place information or review, the write servers add an entry in the relational database and put all the related images in the blob storage.

Making segments: The segment's producer splits the world map taken from the third-party map service into smaller segments. The places inside each segment are stored in a key-value store. Even though this is a one-time job, this process is repeated periodically for newer segments and places. Since the probability of new places being added is low, we update our segments every month.

Detailed design

Quad tree

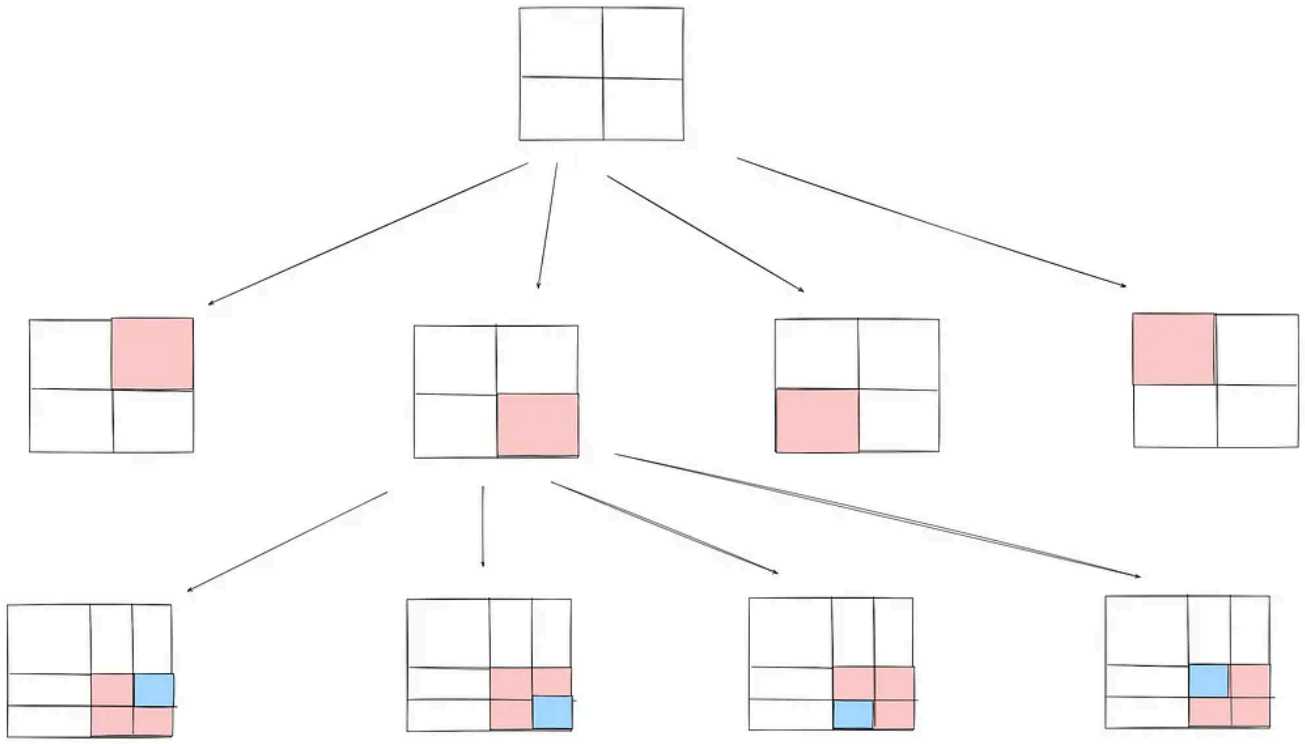

Fig 3.0: QuadTree data structure

As the name suggest, it is 4 node based search tree just like binary search tree. Each node has four children except leaf nodes, and leaf node represent the segment.

We start searching from the root node and continue to visit the nodes to find our desired segment. We check every node to see if it has more child nodes. If a node has no more children, then we stop our search because that node is the required one.

We also connect each child node with its neighboring nodes with a doubly-linked list. All the child nodes of all the parents nodes are connected through the **doubly-linked list**. This list allows us to find the neighboring segments when we can move forward and backward as per our requirement. We explore more nodes until we

reach our search radius, after finding the node, we query the database for information related to the places and return the desired ones using `placeID`.

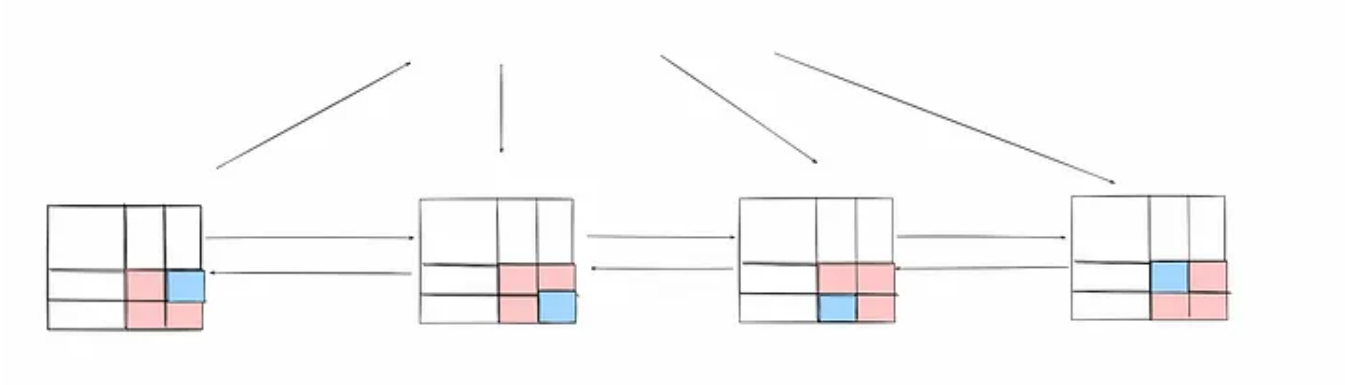

Fig 4.0: Leaf nodes of quadTree are joined by doubly linked list

We replicate our QuadTrees on multiple servers to ensure availability. This allows us to distribute the read traffic and decrease the response time. QuadTrees are built on a server, so we can use the server ID as a key to identify the server on which the QuadTree is present. The value is the list of places that the QuadTree holds. The key-value store eases the rebuilding of the QuadTree in case we lose it.

We insert a new place into the database as well as in our QuadTree. We find the segment of the new place if the Quadtree is distributed on different servers, and then we add it to that segment. We split the segment if required and update the QuadTree accordingly.

We need a service, a **rating calculator**, which calculates the overall rating of a service. We can store the rating of a place in the database and also in the QuadTree, along with the ID, latitude, and longitude of the place. The QuadTree returns the top 50 or 100 popular places within the given radius. The **aggregator service** determines the actual top places and returns them to the user.

Evaluation

- **Availability**: We partitioned the data into smaller segments instead of having to deal with a huge dataset consisting of all the places on the world map. This made our system highly available. We also **replicated the QuadTrees** data using key-value stores to ensure availability.

- **Scalability**: We split the whole world into smaller dynamic segments. This allows us to search for a place within a specific radius and shorten our search area. We then used QuadTrees in which each child node holds a single segment.

Upon adding or removing a place, we can restructure our QuadTrees. So, we were able to make our system scalable.

- **Performance:** We reduced the latency by using caches. We cached all the famous and popular places, so request time was minimized.
- **Consistency:** The users have a consistent view of the data regarding places, reviews, and photos because we used reliable and fault-tolerant databases like key-value stores and relational databases.

System Design Interview | Software Architecture | Software Architect | Yelp Design | Distributed Systems

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium